EXHIBIT E

From: Brian Petry [bpetry@astutenetworks.com]
Sent: Tuesday, September 17, 2002 12:25 PM
To: patents
Subject: Info for provisional patent #3: Programmable context (flow state) area for the application
Re: Provisional patent kick-off for:

"Programmable context (flow state) area for the application"

From: Brian Petry

High-Level Design (HLD) References:
[1] Dispatcher: Search for interface core, "dual core mode" 2.2.2
[2] FDC: "dual core mode" 1.2; search for "dual mode" and "dual core mode"
[3] Lookup Controller: 1.2.5, 2.3.5.3, 2.3.5.4; Split workspace w/ sharing: 2.3.5.4

One aspect of our architecture is the ability of an application to use the flow workspace flexibly. We want to define the application in two parts: a lower-level protocol and an upper-level protocol. For example, TCP can be a lower-level protocol and the upper-level can be HTTP or iSCSI. I think the original intent of this disclosure was to just focus on upper-level part as the "application," but I think we should claim the inventions involving both the lower-part and the upper-part.

Our hardware processing elements have features that enable different CPUs to process the lower-part and the upper-part. This is called "dual core" mode or just "dual mode" by the dispatcher and FDC. It's called "split workspace" by the LUC. The goal of these features is to reduce the amount of state lookups and write-backs between the chip and external memory. In dual-core mode, it is assumed that an incoming event needs to be processed by both a protocol core (the lower part) and an interface core (the upper part). In a standard protocol stack diagram that you may have seen, protocols are often divided into layers. Here, two adjacent layers can correspond to the lower part and upper part of an application. Our chip divides processing resources between the lower part and upper part with dual-core mode.

The benefits of dual core mode is in 3 parts:

1) The upper- and lower-parts of the flow state are stored in the same contiguous block of memory, with a logical "split" between them (the memory offset of the split point is configurable). The block of memory is retrieved from DDR and written to DDR at once rather than separately. Without dual-core mode, the application context would need to be retrieved and stored separately.

2) Also, splitting the processing between cores can provide some savings of processing resources: CPU time and local memory (per-core memory). If the processing split is arranged correctly, the protocol core can do some processing and then hand-off to the interface core. After handing-off, the protocol core can go on to start processing the next event in its queue. This is in the fashion of a protocol processing "pipeline," the first stage being the "lower part" and the next stage the "upper part."

3) Without dual-core mode, the protocol core would have to "chain" to the next stage of processing, the "upper part" of an interface core by sending an event to the dispatcher. That event would be "stateful" because the interface core needs its own state. Note that some applications would benefit from this "chaining," and that feature is also an invention in a different disclosure. But some applications may be hindered by the additional overhead imposed by chaining.

Note this limitation of the chip architecture: once workspaces (flow states) are delivered to a core, they cannot be moved around or reassigned to other cores. Currently, we have no plans for a design that supports workspace movement. Do you think we should try to cover this in case our competition decides to use that technique instead of something like dual-core mode?

As currently defined by the HLDs, dual-core mode is a global mode. That is, when dual-core mode is enabled, all stateful events are allocated a protocol core and an interface core. But the invention should not be limited to this. It is reasonable to have dual-core mode selected in the dispatcher by event type or possibly in the LUC based on a field in a flow state header. This per-event, or per-flow dual-core selection would be more flexible---enabling multiple protocol stacks, for instance one requiring dual-core mode and another not, to run on the chip at the same time.

The first processing element to deal with dual-core mode is the dispatcher via the FDC. In dual-core mode, the dispatcher and FDC have to allocate 2 event slots and 2 workspace slots on 2 cores: a protocol core and an interface core. The next element is the LUC. It moves the flow state into processors' workspaces according to the programmed split. Note that the HLD defines the split offset a global parameter. As with dual-core mode, the split offset could be made per-event or per-core selectable in a different embodiment. Next is the protocol core. After processing the event and sending the workspace write-back command, it sends an inter-core (intra-cluster) event to the interface core. Note that this event has less chip overhead than a dispatcher event. The interface core completes its processing and issues a write-back. The dispatcher can then free-up the

event and workspace slots for the flow and the LUC can write-back both the upper- and lower-parts of the flow state as one contiguous chunk.

A related feature to dual core mode is selective workspace write-backs. We will probably want to cover this feature somewhere. This disclosure may be the right place. Once a workspace is delivered to a core, the software on the core can decide which parts of the workspace need to be written back to external memory. This is like the "dirty bits" of a processor data cache that keep track of which cache lines the processor has written-to. In a different embodiment, hardware can keep track of "dirty" sections of a workspace to automatically write-back the dirty parts without software keeping track.